

Evolution in the Core

A journey through the basics of artificial life

Project leader: Nenad Tomašev

Participants: Blanca Gonzalez Bermudez, Ivan Matej Kolobarić, Toni Marković

I. Introduction

Evolution in general

“It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.” Charles Darwin

Evolution is all about adaptation. As the environment changes its inhabitants have to adapt if they want to survive. The question is: How do living organisms adapt to environmental changes? The survival potential of a living organism is contained within its genetic code which is contained in the genotype. Phenotype emerges as a product of an interaction between the genotype and the local biotic and a biotic environment. Genotype in a species changes over time as a result of mutations under evolutionary pressure. This process takes place over many generations, and is one of the basic phenomena of life as we know it.

Evolution in computer science

Evolution has inspired computer scientists by giving them new and powerful tools to use in improving programs that they make. Evolution basically optimizes the performance of organisms in the environment in the same way that living organisms become more adapted to the world they live in. Computer programs are usually made in order to solve certain tasks. To be able to solve their tasks more efficiently they must be optimized. It is possible to use evolution to address this problem.

CoreWar history

CoreWar was inspired by 2 predator programs Creeper and his sequential program called Reaper that destroyed copies of the Creeper. CoreWar was first mentioned in 1984 Scientific American article by A. K. Dewdney.

CoreWar basics

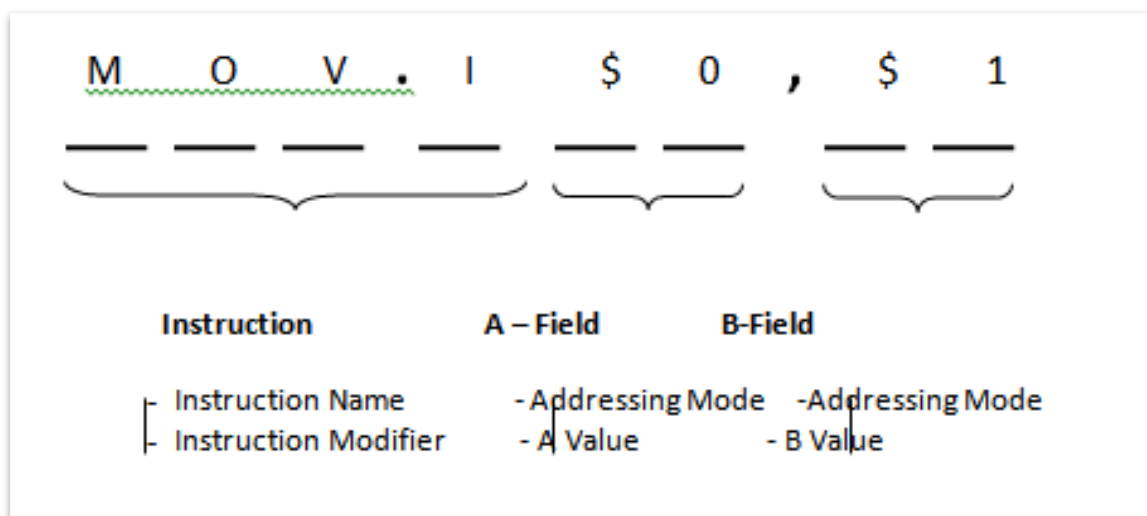
In CoreWar, programs written in an assembly-like language called Redcode compete against each other in a simulated environment referred to as the *core* that is being controlled by MARS(Memory Array Redcode Simulator). These programs are called warriors. They are loaded into the core at random positions and begin executing their instructions sequentially. They control their execution by creating new threads or making loops where some strategy is used iteratively throughout the simulation. A warrior wins when his opponent has no more active threads. A thread is eliminated when it executes a DAT instruction or division by zero.

RedCode instructions

Redcode syntax consists of 19 instructions, 7 instruction modifiers and 8 addressing modes. Each command consists of an instruction name, followed by the instruction modifier, A-field addressing mode, A-field value, B-field addressing mode and B-field value. A-field is the source and B-field is the destination of the operation performed by the instruction.

Structure of a Redcode Instruction:

Example 1:



DAT removes the thread that executes it from the thread queue. It is used to store data. MOV copies the source to the destination. Depending on the modifiers, it can copy either the instruction field or the whole instruction.

Example 1 shows the simplest warrior - the imp. It copies the whole instruction, not just the field values. The relative addressing mode \$ is used in both fields. The instruction copies itself one field down the core array.

CoreWar warrior types

- 1) Replicator (paper) - Type of warrior that makes repeated copies of itself and executes them in parallel so that the copies start making copies. Its goal is to fill the whole core with its instructions. Replicators are tough to kill, so they usually score many draws.
- 2) Bomber (rock, stone) - Blindly copies "bombs"(made of DAT instructions) to locations in the Core hoping to hit the enemy instructions.
- 3) Scanner (scissor) - Scanner doesn't attack blindly but tries to locate his opponent before engaging it.
- 4) Imp (crawler) – Simple warrior that crawls through the core. Imps aren't very useful in killing opponents but provide the warrior with robustness and durability. This type of warrior was named after the first warrior ever made by A.K. Dewdney..
- 5) Core Clear – is a simple warrior that sequentially overwrites every instruction in the core, sometimes even including itself. Core clears are not very common as stand-alone warriors, but they are often used as an end-game strategy by bombers and scanners.

Hybrid strategies

The warriors nicknames remind of a game called "Rock, paper, scissors". They are used to point out the weaknesses of the warrior types in battle versus each other. Paper has advantage over stone, scissors over paper and stones over scissors. Though the game is lacking a lizard and Spock it is still interesting. Hybrids are combinations of 2 or more warrior types. By combining various warriors you decrease some warrior types weaknesses to other warrior types. In CoreWar there are different strategies in combining warriors. For example you can make stone-imps, stone-papers, and scissors with core clear etc. The best warriors in the world are very sophisticated and combine a lot of strategies.

II. Methodology

General Approach

In order to simulate evolution of warriors in the core, we created a genetic algorithm in java programming language. The program mimics the basic mechanisms of evolution: mutation, recombination, and selection, in order to create better warriors from existing ones. The evolutionary cycle of our algorithm is shown below.

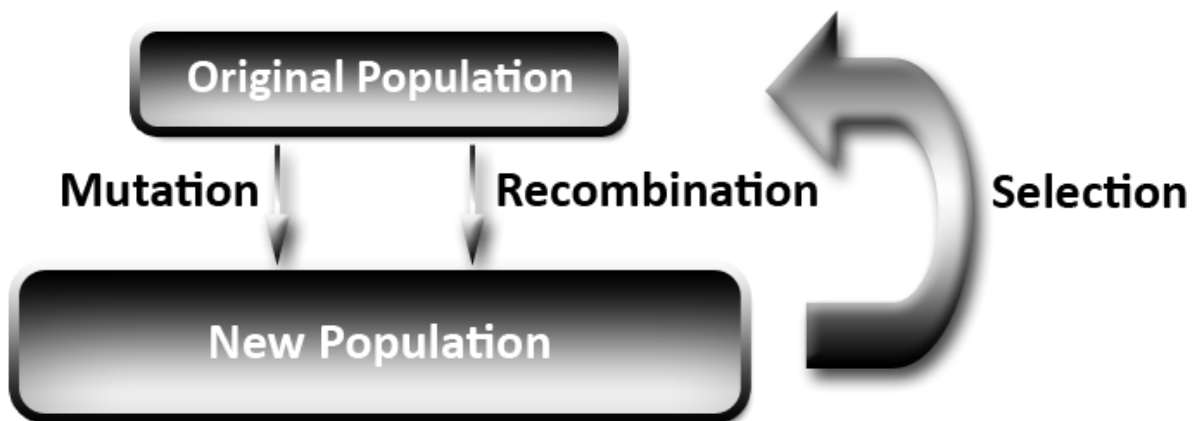


Figure 1 – The evolutionary cycle

From the original population (which can initially be obtained in several different ways), we use mutation and recombination to increase the population to a new size. Then, using a number of battles, each warrior is evaluated and the best ones „survive“ and appear in the next cycle, passing on their „genes“ and traits.

Starting population

We used two different ways to obtain a starting population. One approach was to select a number of existing human made warriors, taking care that the population is balanced and contains every type of warriors. This approach took more time, because choosing the right warriors is a long process. However, it provides a „genetic pool“ that already contains some complexity and functionality.

The other approach was to generate the initial population automatically by the computer. This means that the program created warriors by combining possible instructions and numerical values. Although that process was random, we made some values more probable to appear, because some basic instructions need to appear more often in order to make functional warriors, while some instructions are very rarely useful so we wanted to avoid them. This approach would lead to a very simple and mostly useless initial population which has a potential to develop into something more complex.

Mutations

Mutations were the mechanism that we used in order to create variety and possibly improve existing warriors. It was implemented in two ways that differ in their complexity. The first approach was quite simple: when a warrior is selected for mutation, the program goes through its genes and randomly determines if each single gene will be mutated. If yes, each part of the gene (instruction, instruction mode, A and B values and modifiers) has a certain probability to be changed. Just like when creating warriors, some instructions have a higher probability of being used than others. Also, the start point (which denotes the line from which the code starts executing) can be changed randomly.

The second approach was similar, but it introduced a different probability distribution for new instructions, which is different depending on what gene is changed. For example, if a gene containing the JMP instruction is chosen to be mutated, the most probable instructions to replace JMP would be JMZ, JMN or DJN, because they are all essentially variations of the jump instruction. Changing JMP to another type of jump instruction might be more beneficial than changing it to, for example, ADD instruction, which has a completely different function. The same was applied to instruction modes, and to AB values and modifiers.

Recombination

We implemented recombination in the following way: a piece of code of random length (between 30% and 70% of the length of the whole warrior) at a random position from Warrior A is exchanged with a piece of same length at a different random position on Warrior B. As the results of recombination, two warriors are created, both containing parts of warriors A and B. It can be represented with the following diagram:

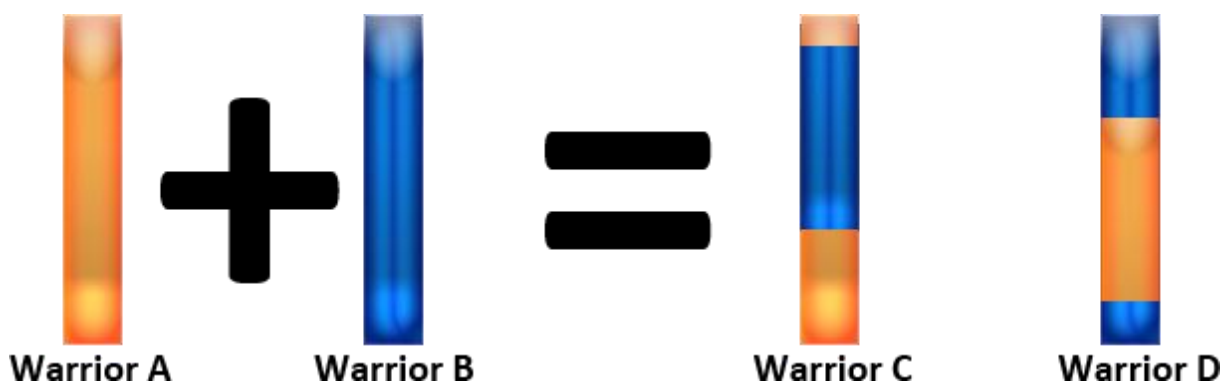


Figure 2 – Recombination

We had two different approaches to the selection of parents. One was completely random, while the other used a fitness function that measures the fitness of each warrior to determine the probability of it becoming a parent to recombinant warriors. A series of battles are done by choosing random pairs of warriors to fight. 25% of all possible pairs are run, while no battles are repeated and a

warrior cannot fight itself. The algorithm ensures that each warrior fights at least once. A score is tracked for each warrior and the average score of all his battles is used as its fitness. A warrior with the maximum average score of 3 would have a 3 times larger chance of being a parent than a warrior with the average score of 1. This makes sure that the good traits that help warriors to obtain a higher score are passed on and combined with other good traits.

Selection

The final stage of the cycle is selection of warriors who will enter the next cycles. For this part we also had some different methods. One was to fight all of the warriors with a set of human made warriors (the benchmark) and calculate their scores, after which the best warriors are selected and the rest are discarded. The benefits of this approach are that it provides a quite accurate representation of the strength of the warriors, which can also be compared between different generations because the benchmark stays the same. While the drawbacks are that it takes a large amount of time to simulate all the battles. Also, the warriors in the benchmark must be carefully selected to include both weak and strong warriors of different types.

The second technique was to divide the existing warriors into groups randomly and have them fight among their groups. The best third of warriors in each group passes onto the next cycle, while the rest are discarded. A big advantage of this system is that it takes less time to do battle simulations, but the results might not be as accurate as when running all the battles. Also, this makes it impossible to compare the results of different generations, because they all fight different opponents.

III. Results

Standard core size warriors

This approach showed quite a lot of success. It was started by generating a random initial population of warriors. Mutations were carried out by the simple approach with only one probability distribution, and the fitness function was used to determine the probability of each warrior becoming a parent in recombination. Selection was carried out with the group system.

In 61 evolution cycles it was easy to observe the evolution of warriors within the core. It started with simple warriors that were random but good enough to survive. It continued with the simplest warriors who were able to core-clear. Later different types such as replicators and core-clear/stone hybrids emerged. Clearly, each generation brought a slight improvement to the performance, so the core-clearers from the last generation were significantly more efficient than the first ones that appeared. However, we were unable to extend this simulation longer to observe further developments, due to the limited time for the project.

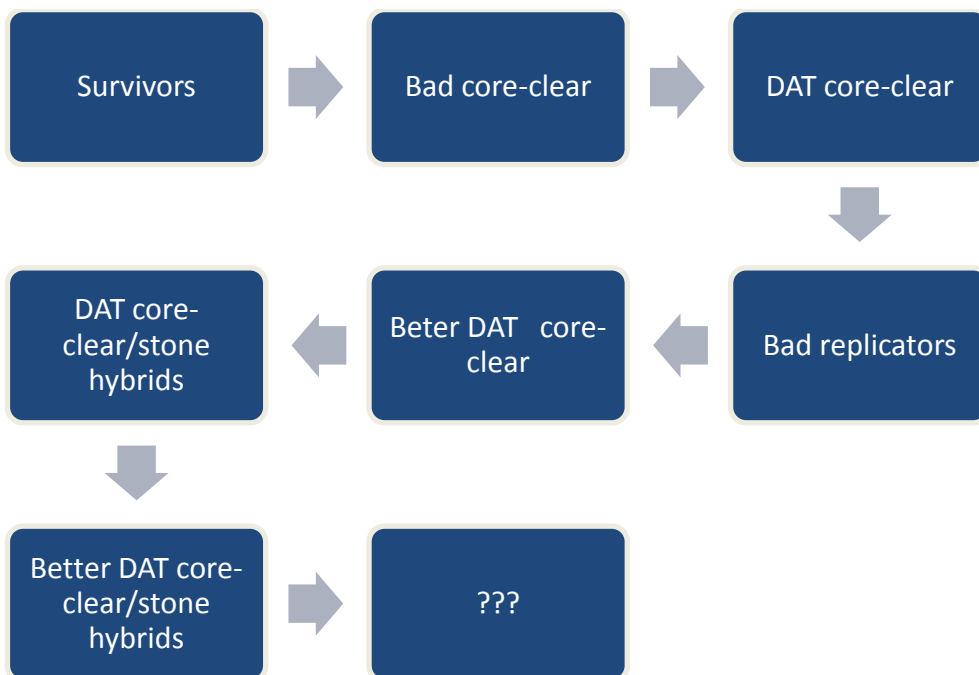


Figure 3 – The development of different types of warriors through evolution

Nano warriors

Apart from warriors made for standard core size, we also tried evolving so called Nano warriors, which are limited to 5 lines in length and compete in a smaller core. Due to battles of Nano warriors being much shorter than standard battles, we managed to run more than 140 evolutionary cycles.

Our approach to this version was in different in the fact that we adjusted the probability distribution for mutations of instructions to suit the needs of Nano warriors – since they are much shorter, they cannot contain complex code, and the most successful types are simple one-shots and core-clearers. Therefore, we increased the probabilities for MOV, SPL and JMP instructions, which are most commonly used in these conditions.

The result of this approach can be seen in the results of the best warrior of each Nano evolution cycle against the KOTH Nano Hill that was used as a sort of a benchmark. The general upward trend in the graph below is obvious. The occasional drops in score can be explained by the imperfect system of determining the best warrior in a generation (only 25% of battle pairs are processed to save time).

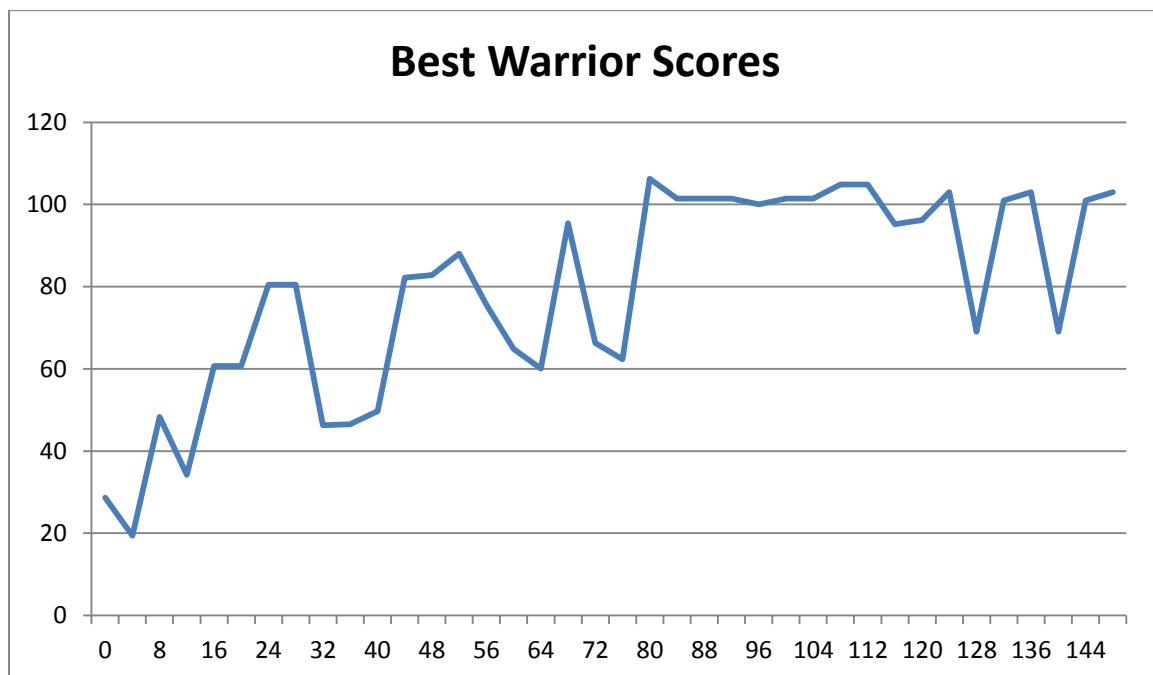


Figure 4 – best warrior in each generation against its score on KOTH Nano hill.

Experimentation

Experiments relating to the other approaches with a different population initialization mechanism and different mutations were also performed. Due to technical difficulties, it wasn't possible to run as many generations of the algorithm as was previously conceived. Therefore, we can only draw some basic conclusions from the partial results of the experiments. These approaches also led to generating working warriors in a few generations and the rapid drop of fitness from changing human-coded warriors by random mutations in initialization process started slowly decreasing in time. Naturally, with time being the keyword, it wasn't possible to actually check if the evolved warriors would be able to reach the level of warriors used for seeding the initial warrior pool. As in the first iterations of the only approach for which we managed to run enough iterations, mutation-resistant forms of artificial organisms were the first to emerge, more specifically core-clears and replicators.

IV. Conclusion

We have shown that evolutionary approach can be successfully applied to automatically generate and improve CoreWar programs. To do that, we made a Java CoreWar evolver which has options of doing mutation, recombination and population initialization in several ways. During the runs of the genetic algorithm, several types of warriors emerged, successively replacing each other as the population evolved and complexity increased. Mutation resistant forms were the most frequent ones, since they were not as likely to die from small random changes to their executing code. It is difficult to analyze such results, because a large amount of CoreWar warriors was generated by the evolver. More complex strategies didn't appear by the end of the experiments, because usually more generations are needed to achieve higher complexity.

V. Acknowledgements

It is our pleasure to express our gratitude to our project leader Nenad Tomašev. Thank you for putting up with us and our mistakes, for the patience you have shown to computer science beginners, and for all the things you have taught us. We also want to thank The Summer school of science for providing us the opportunity to participate in a scientific project, to meet wonderful people and above all to learn the various paths of science.

References

- Building a CoreWar Optimizer – Nenad Tomašev, Novi Sad 2008
- http://en.wikipedia.org/wiki/Core_War
- <http://java.sun.com/>
- <http://sal.math.ualberta.ca/hill.php?key=nano>